Képfeldolgozó algoritmusok optimalizálása[1]

Max Gyula

Budapest Műszaki és Gazdaságtudományi Egyetem,
Automatizálási és Alkalmazott Informatikai Tanszék
1521 Budapest, Pf. 91, e-mail: max@aut.bme.hu

Abstract: A forgalom megfigyelő rendszerek (Traffic Observation System [TOS]) egyik központi modulját a képfeldolgozó függvények alkotják. A modulban található függvények gyorsasága alapvetően meghatározza a rendszer gyorsaságát. Ezek a függvények jelentős mértékben hozzájárulnak ahhoz, hogy az adott rendszert real-time üzemmódban tudjuk-e működtetni. A forgalom azonosításához szükséges alapvető függvények, mint pl. az élkeresés vagy a háttér leválasztás, eléggé erőforrás igényes művelet. Ennek megfelelően, ha ezek erőforrásigényének nagyságát csökkenteni tudjuk, akkor azzal akár jelentős sebesség növekedést érhetünk el a képfeldolgozásban. A cikkben a mai x64-es processzorokra írt rendszerek gyorsaságának növelésével foglalkozunk, bemutatva a magasszintű nyelvek és a gépközeli utasításokban rejlő sebességnövelési lehetőségeket. Bemutatjuk, hogy az x64-es processzorok és a hozzájuk tartozó assembler szintű nyelvek hogyan gyorsíthatják fel rendszereinket..

1. BEVEZETÉS

Az utak mentén egyre több helyen találkozhatunk térfigyelő rendszerekkel. Ezek egy része már nem csak egy human megfigyelő rendszer része, hanem önálló működésre és esetleg döntéshozatalra is képes eszköz lehet. Ezeknek az eszközöknek alapvető jellegzetessége a gyorsaság, hiszen real-time módon kell reagálniuk a környezet változásaira, ellenkező esetben ugyan nem tudnának adekvát válaszokat és döntések adni a körülöttük lejátszódó folyamatokra.

Többször megfogalmazódott már az az igény, hogy hogyan lehetne a rendelkezésre álló közterületet olyan mértékben és gyorsasággal monitorozni, hogy a rendelkezésre álló adatok alapján valós idejű forgalmi előrejelzéseket lehessen adni a közlekedésben résztvevők számára. A képfeldolgozó technikák javulása és a mikroprocesszorok teljesítményének jelentős emelkedés együttesen teszi gyorsabbá és pontosabbá a stream alapú forgalomi paraméterek real-time képfeldolgozását [Int06]. A környezetünkben lejátszódó folyamatokat a forgalom megfigyelő rendszerekben kialakított modelleken keresztül vizsgáljuk, melyekben számos lényeges elhanyagolás mellett a vizsgálni kívánt folyamat, számunkra érdekes részletére figyelhetünk. Ennek megfelelően az egyes moduloknak különböző feladatuk van. Míg egyesek a jármű fizikai paramétereit azonosítják vagy megfigyelik pályaadataikat, addig mások az időjárás változásainak vagy a rendellenesen mozgó járművek kiszűrésével foglalkoznak. Azonban mindegyik technika a képfelismerésen alapul. A beérkező képek feldolgozása általában valamilyen zajszűrő beiktatásával kezdődik. A módszer lehetőséget ad arra, hogy a szűrés után létrejövő felvétel egyes képkockáit vagy annak részleteit könnyebben össze lehessen hasonlítani. Ezzel egyszerűbbé válik a háttérszűrés vagy a forgalmi területek kiválasztása. Mindezek elősegítik a forgalmi paraméterek, pl. a forgalmi sávok vagy szabálytalanságok detektálását [Max, (2012)], a vizsgált területen egységnyi idő alatt áthaladó járművek számának meghatározását, vagy az áthaladó járművek sebességének mérését.

2. KAPCSOLÓDÓ CIKKEK

A forgalom megfigyelő rendszerek képfeldolgozási stratégiái éveken át a mozgó járművek felismerésére, azonosítására [Max, (2014)] vagy követésére törekedtek [Kun, (2009)]. Nagyon fontos tényező volt a képfeldolgozás valós időben történő elvégzése. A cikkekben leírt valós idejű rendszerek többféle funkciót láttak el, az egyes sávok lezáródásától kezdve a jármű követésen keresztül, a járművek forgalmi dugókba történő érkezéséig [Kanhere, (2007)]. Annak érdekében, hogy a fentebbi forgalmi helyzeteket valós időben lehessen kezelni, megfelelően gyors hardverre és szoftverre van szükségünk. A megfelelően megírt szoftverek elkészítéséhez két utat használhatunk. Vagy már egy elkészített szoftvercsomagot használunk, vagy megírjuk magunk a szükséges rutinokat. Időhiány miatt, általában az elsőt választjuk. legtöbbször az OpenCV-re esik a választás [Uke, (2012)]. Az OpenCV - Open source Computer Vision – mint neve is mutatja egy nyílt forráskódú könyvtár, real-time folyamatok feldolgozásához [Int01, Int07]. A többmagos processzorokra is megírt C és C++ nyelvű alapkönyvtár több mint 500 beépített függvényt és több mint 2500 optimalizált eljárást tartalmaz [Int02]. A könyvtárban található eljárások zöme általános funkciók megvalósítására használható, ami azt jelenti, hogy mind az adat előkészítő, mind pedig az

1. This work was partially supported by the TÁMOP-4.2.1.D-15/1/KONV-2015-0008 project.

adatfeldolgozó része számos olyan részt is tartalmazhat, amit adott esetben a felhasználó nem használ ki [Int03, Int04]. Ezek kiküszöbölésére és a gyorsaság növelése érdekében, a nyílt forráskód miatt lehetőségünk van a nem kívánt részek törlésére. Ha azonban az eredmény így sem felel meg a várakozásainknak, akkor már csak a programnyelv változtatás lehetősége marad számunkra. A későbbiekben látni fogjuk, hogy a C vagy a C++ meglehetősen jó gépi kódot generál, azonban ezek a programok is az általános felhasználhatóság érdekében íródtak. Ebben a cikk bemutatunk néhány példát és ellenpéldát arra nézve, hogy milyen gyorsítási lehetőségeink vannak, ha nem magas szintű nyelvet, hanem gépközeli assembly programozást választunk [Int05, Int08, Int09, Int10].

3. SIMD TECHNOLÓGIÁK

A képfeldolgozó eljárásokat széles körben használják, melyek sok, alapvetően megegyező jellemzőket mutatnak. Többnyire 8 vagy 24 bites pixelek vagy 16 bites audio minták feldolgozása történik meg. A feldolgozás során több, gyakran ismétlődő ciklust használunk, hiszen az egyes képelemek összehasonlítása vagy megkeresése a feladatunk. Ezen ciklusok hossza általában nem túl nagy, mégis számos sok közös vonást mutat. Egyrészt elmondható róluk, hogy erősen erőforrás igényes műveletek. Sok bennük a szorzás vagy az összeadás, de ennél bonyolultabb művelet többnyire csak ritkán szerepel a főbb ciklusokban. Másrészt az is nyilvánvaló, hogy feladataink egy része erősen párhuzamos adatfeldolgozást igényel, hiszen egy kép összehasonlítás folyamán általában a kép ugyanazon pontjait hasonlítjuk össze, és ehhez az azonosításhoz legfeljebb a képpont környezetének ismeretére van szükség, de nem kell hozzá az összes többi pixel ismerete. Vagyis ugyanazt a műveletet vagy műveletsort kell minden egyes adaton végrehajtani. Ezek a SIMD - single instruction, multiple data – műveletek a párhuzamosan kapcsolt számítógépek egy osztályát, illetve a párhuzamosan végrehajtható utasítások vagy utasításkészletek egy csoportját jelölik. A SIMD utasítások lényege, hogy egy utasítás egy nagyobb adathalmazon végzi el ugyanazt a műveletet, pl. egész típusú adatok vektorán. A SIMD gépek az adatszintű párhuzamosságot használják ki. Ezért a SIMD utasítások igen hatékonyan alkalmazhatók különféle digitális (audio, video és egyéb szignál) jelfeldolgozási feladatoknál.

3.1 MMX architektúrák

Maga az MMX elnevezés több jelentéssel is bír. Alapvetően a MultiMedia eXtension rövidítése lenne, de emellett még használják a Multiple Math eXtension, vagy a Matrix Math eXtension kifejezést is. Az MMX technológia középpontjában az áll, hogy gyorsítsa a multimédia, a kommunikáció és a numerikus intenzív alkalmazások kihasználásához a párhuzamosságban rejlő multimédiás és kommunikációs algoritmusok gyorsaságát, miközben fenntartja a teljes kompatibilitást a meglévő operációs rendszerrel és az alkalmazásokkal. MMX technológia 1997-től lehetővé tette, hogy a Pentium processzor kezelni tudja a multimédiás feladatokat anélkül, hogy költséges digitális jelprocesszorokat (DSP) kellene alkalmaznunk. Előnyeit a következőkben foglalhatjuk össze:

• Pakolt adattípusokat használ - kis adatelemek csomagol össze egy regiszterbe

• 57 új, továbbfejlesztett utasítást tartalmaz, amelyek az összes adatelemen, egy regiszterben, párhuzamosan, SIMD mintára képesek műveleteket végrehajtani

• 8 db 64-bites MMX regiszterek, MM0-tól MM7-ig, hogy a regiszterek megfeleljenek az Intel Architecture (IA) lebegőpontos követelményeinek, ahol a 80 bites lebegőpontos regiszterek alsó 64 bitje felel meg az MMX regisztereknek

• Teljesen IA kompatibilis

• Csökkenti a multimédiás rendszerek költségeit

Az előnyök mellett jelentős hátránya is volt az MMX műveleteknek, mivel utasításait csak egész számokon tudta végrehajtani.

A teljesség igény nélkül bemutatunk néhány MMX utasítást, amelyek jól használhatók a képfeldolgozás során:

PMADD[WD]: Pakolt szorzás és összeadás

A művelet két MMX regiszterben elhelyezett 4-4 16 bites egész szám páronkénti 32 bites szorzatát állítja elő, majd az MMX regiszter alsó és felső bitjein található szorzatokat az 1. ábra szerint összeadja. A művelet jól használható pl. mátrix szorzások esetében. Az 1. ábrán bemutatott példa négy szóból két duplaszavas eredményt állít elő.

I1	I2	I3	I4
*	*	*	*
K1	K2	K3	K4

I1*K1+I2*K2	I3*K3+I4*K4

1. ábra: PMADDWD 16 bites szavak szorzásával és ezek összeadásával 32 bites eredményt állít elő

PCMPGT[W]: Párhuzamos, szavas összehasonítás

A 2. ábrán két 16 bites egész szám összehasonlítását láthatjuk 4-szer.

20	45	14	0
>	>	>	>
42	12	14	14

0000h	FFFFh	0000h	0000h

2. ábra: PCMPGTW 4 szó páronkénti összehasonlítása

Ez az MMX művelet jól használható pl. küszöbérték meghatározásoknál.

3.2 SSE architektúrák

A következő lépést az 1999-ben bevezetett Streaming SIMD Extension (SSE) jelentette. A 70 utasítást tartalmazó SSE

már a duplapontosságú lebegőpontos adattípusokat is bevonja a vektoros adatfeldolgozásba, bár többségük csak egyszeres pontosságú lebegőpontos adatokon dolgozik. Az SSE eltér az IA-32 architektúra MMX utasításkészletétől: külön regiszterkészletet használ (az XMM regisztereket), és néhány új integer utasítást tartalmaz, amik az MMX regisztereket használják. Az SSE lebegőpontos utasításai az XMM regiszterekkel kényelmesebben használhatóak, mint az FPU verem-alapú "regiszteres" megközelítése. Tipikusan olyan helyeken használják, ahol mindig pontosan ugyanazokat a műveleteket kell végrehajtani adat-objektumok sokaságán. Az SSE család (a SIMD utasításokon kívül), tartalmaz még néhány gyorsítótár-kezeléssel kapcsolatos utasítást is. Ezekkel lehet szabályozni az L1/L2 gyorsítótár működését. 2000-től az optimalizált Pentium 4 processzor már kitűnően hasznosítja a gyors memória-alrendszerben és az SSE2 utasításkészletben rejlő teljesítménynövelő jellemzőket. A Pentium 4-től, az erre az architektúrára épülő Celeron processzorokban megtalálható SSE2 utasítások javítják a multimédiás és internetes alkalmazások, valamint a játékszoftverek futtatásának képességét.

• 128 bites vektor regiszterek

• Támogatja mind az egyszeres, mind a duplapontosságú lebegőpontos műveleteket

• Több újabb verziója is született: SSE3, SSSE3, SSE4.1, SSE4.2

Bemutatunk néhány érdekes műveletet az SSE utasítások közül is.

Az utasításoknak két típusa van:

• Skalár (Single scalar perfix) csak a legkisebb helyi értékű 32 bites duplaszón végzi el a műveletet a 3. ábra szerint

mulss xmm1,xmm0

XMM0	20	45	14	4
				*
XMM1	42	12	14	32
XMM1	20	45	14	128

3. ábra: Skalár szorzás xmm regiszterek esetén

• Párhuzamos szorzás pakolt duplaszavak esetén ps (parallel scalar prefixszel), melyet a 4. ábra mutat be.

mulps xmm1,xmm0

XMM0	20	45	14	4
	*	*	*	*
XMM1	42	12	14	32
XMM1	840	540	196	128

4. ábra: Párhuzamos skalár szorzás xmm regiszterek esetén

Az SSE műveletek között az SSE3-tól kezdve találhatunk aszimmetrikus műveleteket is.

ADDSUBPD: Két XMM regiszter felső helyi értékein elhelyezkedő float számokat összeadja, míg alsó helyi értékű float-okat az 5. ábra szerint kivonja egymásból.

ADDSUBPD xmm1, xmm0

XMM0	Y0	X0
	+	-
XMM1	Y1	X1
XMM1	Y0+Y1	X0-X1

5. ábra: Aszimmetrikus utasítás

3.3 AVX architektúrák

Az AVX (Advanced Vector Extensions) egy az Intel Sandy Bridge mikroarchitektúrával 2011-ben bevezetett SIMD utasításkészlet, amely 256-bit széles utasítások feldolgozását teszi lehetővé. Az AVX alapvetően a lebegőpontos feldolgozás gyorsítására lett kitalálva, és az egyik legérdekesebb újítása, hogy lehetővé teszi a háromoperandusos műveletvégzést, tehát az a:=a+b helyett immár használhatjuk a c:=a+b formát is (igaz, megkötésekkel). Az eredmény már nem kötelezően írja felül a bemenő adatokat, célként megadható más terület is. Általánosan [c1, c2, ...]:=[a1, a2, ...] operandus [b1, b2, ...] formában írhatóak fel, így ha a bemenő értékekre szükség van a későbbiekben is további számításokhoz, megspórolható egy adatmásolás. Az AVX jól alkalmazható jelfeldolgozásnál (signal processing), különféle multimédiás területeken, vagy tudományos szimulációk esetén.
Az AVX2 (vagy 2.0) az Intel Haswell mikroarchitektúrájával debütált. Ez az utasításkészlet az egész számos SIMD végrehajtást is kibővítette a korábbi 128-ról 256 bites vektorokra, leváltva/kiegészítve a koros 128 bites, Pentium 4-gyel bemutatkozó SSE2 készletet, valamint a Core 2 CPU-kban megjelent, ugyancsak 128 bites SSE4.1 egész számos SIMD utasításainak nagy részét.

CVTPD2DQ xmm1, xmm0/m128: a 6. ábrán a dupla pontosságú pakolt számokat konvertáljuk duplaszavas egész számokká

YMM0	20.0	45.7	14.6	4.0
YMM1	0	20	45	14	4

6. ábra: Duplapontosságú szám egész számmá alakítása

Ezt a műveletet kicsinyítés/nagyítás esetén használhatjuk.

VBROADCASTSD ymm1, m64: egy dupla pontosságú 64 bites számmal a 7. ábrán feltölt egy 256 bites regisztert

m64	F2h

YMM1	F2h	F2h	F2h	F2h

7. ábra: Adatsokszorozás, adatfeltöltés

4. OPTIMALIZÁLÁSI LEHETŐSÉGEK

A következőkben néhány egyszerű példa keretében bemutatjuk azokat a lehetőségeket, amelyek segítségével a magasszintű programnyelveken megírt programokat érdemes, a gyorsaság növelése érdekében, gépi kódra cserélni. A lehetőségeket néhány egyszerű példán keresztül mutatjuk be, valamint a Függelékben egy komplett példát is mellékelünk.

4.1 Képek összeadása

Két képet szeretnénk összeadni. A képeket tároljuk egy-egy vektorban. Értsük az összeadást szó szerint, azaz az eredmény képünk a két input kép összegeként fog megjelenni.

C-ben megírva a 8. ábrán látható programot, semmi különlegességet nem tapasztalunk. Az egyetlen dologra kell csak vigyázni. Ha két egybájtos pixelt összeadunk, az eredmény 255-nél nem lehet nagyobb, amit itt egy egyszerű kasztolással biztosíthatunk.

```
#define LEN nnnn
int main() {
 unsigned char v1[LEN],v2[LEN],v3[LEN];
  int i,j,t;
    for (j=0;j<LEN;j++){
      t=v2[j]+v1[j];
      v3[j]=(unsigned char)(t>255?255:t);
    }

return
}
```

8. ábra: Két kép összeadása C-ben

Látható, hogy ha el akarjuk kerülni a 8 bites számok túlcsordulásának problémáját, meg kell vizsgálnunk, hogy az eredmény benne van-e a kívánt intervallumban, és ha nem akkor a megfelelő intervallum legmagasabb értékére állítjuk az eredményt. Ez a vizsgálat a belső ciklus sebességét igencsak lelassítja, mert a programunkba minden ciklusban belekerül egy feltételes ugrás, ami a pipeline működést erősen lelassítja. Az MMX pakolt szaturált adattípusokkal és a rajtuk végezhető megfelelő matematikai műveletekkel igyekszik ezt elkerülni. Ami azt jelenti, hogy most már a hardver biztosítja, hogy a számunk a megfelelő intervallumban maradjon. Az MMX regiszter struktúrák 4 adatformátumot támogatnak. Egy regiszter lehet egy 64 bites QWORD, vagy két 32 bites DWORD, vagy négy 16 bites WORD, vagy nyolc 8 bites érték. A formátum ezeken belül lehet normál (signed) és lehet szaturált (saturated). Ez utóbbi, szaturált adattípus némi magyarázatot igényel. Arra tervezték, hogy kezelni tudja azokat az eseteket is, amelyek a képfeldolgozás során akkor lépnek fel, amikor egy-egy aritmetikai művelet eredményét a rendelkezésre álló területen nem lehet megjeleníteni túlcsordulás miatt. Ezt a hardverben úgy oldották meg az adott intervallumú egészek közül a változó ilyenkor a legnagyobb értéket kapja eredményül. Egybájtos egészek esetében ez az érték tehát a 255 lesz. A pakolt adatok és a szaturált típusok együttes használata jelentősen megnöveli a kódolás hatékonyságát és sebességét. Hasonlítsuk össze tesztprogramunkat mind a régi 86-os assembly, mind pedig a SIMD utasításokat tartalmaz assembly kóddal. A 9-es ábrán SIMD utasításokat ne, míg a 10. ábrán már az MMX kódokkal ellátott programrészleteket látjuk.

```
ASSEMBLER
.data
  LEN dw len
  kep1 times LEN db 0
  kep2 times LEN db 0
.code
START:
  xor   edx, edx
CIKLUS:
  mov   eax, edx
  lea   ecx, DWORD PTR [esp]
  movzx ecx, BYTE PTR [eax+ecx]
  mov   DWORD PTR [esp+LEN], edi
  lea   edi, DWORD PTR [esp+LEN/3]
  movzx edi, BYTE PTR [eax+edi]
  add   ecx, edi
  cmp   ecx, 255
  mov   edi, DWORD PTR [esp+LEN]
  jle   TOVABB
CSERE:
  mov   ecx, 255
TOVABB:
  inc   edx
  cmp   edx, LEN/3
  mov   DWORD PTR [esp+LEN], edi ;
  lea   edi, DWORD PTR [esp+2*LEN/3]
  mov   BYTE PTR [eax+edi], cl
  mov   edi, DWORD PTR [esp+LEN];
  jl    CIKLUS
```

9. ábra: Két kép összeadása hagyományos assemblerben

Csak a belső ciklus MMX utasítások használata mellett a 10. ábrán egyrészt jelentős egyszerűsödést mutat, mindamellett a kód hatékonysága is jelentősen megváltozott. Mindamellett, hogy a betöltések és a kiírások száma a nyolcadára csökkent,

```
CIKLUS:
      ; nyolc bájt betöltése
  movq mm0,[esi+ebp-LEN]
      ; pakolt unsigned bájtok összeadása
  paddusb mm0,[esi+ebp-2*LEN]
      ; nyolc bájt kiírása
  movq [esi+ebp-3*LEN],mm0
      ; a ciklusszámláló növelése
  add esi,8
      ; ugrás a ciklus elejére
  loop CIKLUS
```

10. ábra: Két kép összeadása MMX utasításokkal

az előző 18 sort eredményesen és áttekinthetően képeztük le 5 sorban. Azaz míg a 9. ábrán 18 sor végrehajtása kellett egyetlen bájton végzett művelet elkészültéhez, addig az MMX-es megoldásban ez 0.625 utasítás bájtonként. Vagyis ez a lefordított kód 29-szer kevesebb utasítással hajtja végre ugyanazt a feladatot.

4.2 Élkeresés

Képfeldolgozás során mindig történik élkeresés. Ezzel a folyamattal tudjuk szétválasztani a vizsgálni kívánt objektumokat, a nem vizsgáltaktól. Élkeresés során az egyik leggyakrabban használt módszer a gradiens operátor, amelynek számos változata létezik. Matematikailag, egy f(x,y) képben, a gradiens nagyságát g(x,y)-nal (1), míg a gradiens irányát θ(x,y)-nal (2) adjuk meg.

$$g(x,y) \approx (\Delta x^2 + \Delta y^2)^{½} \quad (1)$$

és

$$\theta(x,y) \approx \text{arctg}(\Delta y/\Delta x) \quad (2)$$

ahol

$\Delta x = f(x+n,y) - f(x-n,y)$ valamint $\Delta y = f(x,y+n) - f(x,y-n)$

Az n értékének általában egy kis egész számot – általában 1-et – szoktunk választani. Például a gradiens megvalósításának egyik legegyszerűbb módszere az lenne, ha a (3) maszkot használnánk az adott képhez.

$$|\ 1\ |\ 0\ |\ -1|\quad (3)$$

Különböző variációkat használhatunk, mint pl. a Roberts, a Prewitt vagy a Sobel operátorokat. Ezek közül a Sobel a leggyakrabban használt operátor, amely a Gauss-függvényen és annak deriváltjain alapszik. A mi esetünkben, az eredeti 3x1-es maszk méretét kiterjesztjük egy 3x3-as operátorrá (4), amelyben az összes környező pixel hatását is vizsgálni tudjuk. Az x és y maszkok segítségével először konvolváljuk az összehasonlítandó pixeleket, hogy kiszámoljuk a Δx és Δy értékeket, majd ezek felhasználásával a gradiens nagyságát és irányát.

$$S_x = \begin{|1\ 0\ -1|\\|2\ 0\ -2|\\|1\ 0\ -1|\end{} \qquad S_y = \begin{|1\ 2\ 1|\\|0\ 0\ 0|\\|-1\ -2\ -1|\end{} \quad (4)$$

alkalmazva (4)-et (1)-re megkapjuk (5)-öt és (6)-ot.

$$G_x = \begin{|1\ 0\ -1|\\|2\ 0\ -2|\\|1\ 0\ -1|\end{} * \begin{|f(x-1,y-1)\ f(x,y-1)\ f(x+1,y-1)|\\|f(x-1,y)\ f(x,y)\ f(x+1,y)|\\|f(x-1,y+1)\ f(x,y+1)\ f(x+1,y+1)|\end{} \quad (5)$$

$$G_y = \begin{|1\ 2\ 1|\\|0\ 0\ 0|\\|-1\ -2\ -1|\end{} * \begin{|f(x-1,y-1)\ f(x,y-1)\ f(x+1,y-1)|\\|f(x-1,y)\ f(x,y)\ f(x+1,y)|\\|f(x-1,y+1)\ f(x,y+1)\ f(x+1,y+1)|\end{} \quad (6)$$

Ebből a gradiens nagyságát és irányát (1) és (2) alapján már könnyen meghatározhatjuk. A Sobel operátor kevésbé érzékeny az izolált, nagy intenzitású pont variációkra, mivel értékeit három pixel átlagolásából állítja elő. Ezenkívül elfogadható becslést ad az él irányára. A Sobel operátor hátránya, hogy kép szélen nem tud becslést adni, ami szükségessé teszi egy utófeldolgozó használatát is, amely ezekre a területekre az becslést. Bár a Sobel operátor valamivel bonyolultabb, mint a Laplace operátor, mégis viszonylag könnyen megvalósítható hardveresen formában. A Sobel operátor pontossága élek esetében nagyságrendileg 7%-on, míg a szögek esetében 2 fokon belül van, ha figyelmen kívül hagyjuk a kvantálási hibák, az elmozdulás és az elektronikus zajok hatásait.

Ha egy C függvényben akarjuk megírni az élkeresés folyamatát, akkor a 11. ábrához hasonló eredményre juthatunk.

```
#include <math.h>
#define matrix(a,b,c) a[(b)*(cols)+(c)]
void sobel(unsigned char *data, float *output,
           long rows, long cols) {
int r, c;
int gx, gy;
 for ( r = 1; r < rows-1; r++ ) {
  for ( c = 1; c < cols-1; c++ ) {
   gx = -matrix(data,r-1,c-1) + matrix(data,r-1,c+1) +
        -2*matrix(data,r,c-1) + 2*matrix(data,r,c+1) +
        -matrix(data,r+1,c-1) + matrix(data,r+1,c+1);
   gy = -matrix(data,r-1,c-1) - 2*matrix(data,r-1,c)
        - matrix(data,r-1,c+1) +matrix(data,r+1,c-1)
        +2*matrix(data,r+1,c)+ matrix(data,r+1,c+1);
   matrix(output,r,c) = sqrt((float)(gx)*(float)(gx)+
                             (float)(gy)*(float)(gy));
 }
}
```

11. ára: Élkeresés Sobel operátor segítségével

64 bites assembly utasítások alkalmazása esetén 16 darab 8 bites számot tudunk elhelyezni egy XMM regiszterben. A középső 14 értékből 14 Sobel értéket számíthatunk ki. A program betölti a n-1. sort és kiszámítja a Sobel operátor első részeredményét, majd ugyanezt csinálja az n. és az n+1. sorral is. Az részösszegek ismeretében már (1) és (2) értéke számítható. Egyszerre 14 értéket tud beírni a kimenő file-ba. Míg a C-ben megírt kód csak 148 kép Sobel értékeit tudta kiszámítani, addig 12. ábrán bemutatott, x64-es assembly-ben megírt kód 947 képpel végzett, ami 6,4-szeres teljesítmény növekedést jelent. Az assembly kód futtathatósága érdekében a program a C programokhoz hasonlóan a stack-ről veszi fel a paraméterek adatait. Az r8, r9 és r10 regiszterek tartalmazzák az n-1., az n. és az n+1. sor, míg az rsi regiszter a kimenet címét, valamint az .input és az .outpt változók. A 12. ábrán a Sobel operátor egy sora három paraméterének működését mutatjuk be részletesen. A továbbiak is ezen az elven alapulnak.

```
;       sobel ( input, output, rows, cols );
;       char input[rows][cols]
;       float output[rows][cols]
segment .text
global  sobel
```

```
sobel:
; inicializálás
…
      pxor    xmm13, xmm13 ; a munkaregiszterek nullázása
      pxor    xmm14, xmm14
      pxor    xmm15, xmm15
$ujabb_sor:
      mov     rbx, 1            ; az első elemtől kezdünk
$ujabb_oszl:
      movdqu  xmm0, [r8+rbx-1]       ; az n-1. sor
      movdqu  xmm1, xmm0
      movdqu  xmm2, xmm0
      pxor    xmm9, xmm9
      pxor    xmm10, xmm10
      pxor    xmm11, xmm11
      pxor    xmm12, xmm12
      psrldq  xmm1, 1
      psrldq  xmm2, 2
      movdqa  xmm3, xmm0
      movdqa  xmm4, xmm1
      movdqa  xmm5, xmm2
      punpcklbw   xmm3, xmm13
      punpcklbw   xmm4, xmm14
      punpcklbw   xmm5, xmm15        ; az n-1. sor 8 adata
      psubw   xmm11, xmm3
      psubw   xmm9, xmm3
      paddw   xmm11, xmm5
      psubw   xmm9, xmm4
      psubw   xmm9, xmm4
      psubw    xmm9, xmm5     ; az n-1. sor első 8 eredménye
      punpckhbw  xmm0, xmm13
      punpckhbw  xmm1, xmm14
      punpckhbw  xmm2, xmm15
      psubw   xmm12, xmm0
      psubw   xmm10, xmm0
      paddw   xmm12, xmm2
      psubw   xmm10, xmm1
      psubw   xmm10, xmm1
      psubw     xmm10,  xmm2         ;  az  n-1.  sor  utolsó  6
eredménye

      ; az n. sor feldolgozása
      ; az n+1. sor feldolgozása
; a gradiens nagyságának meghatározása
      pmullw  xmm9, xmm9     ; Gx és Gy négyzetei
      pmullw  xmm10, xmm10
      pmullw  xmm11, xmm11
      pmullw  xmm12, xmm12
      paddw   xmm9, xmm11    ; majd ezek összege
      paddw   xmm10, xmm12
      movdqa  xmm1, xmm9
      movdqa  xmm3, xmm10
      punpcklwd xmm9, xmm13     ; A 4 alsó kétbyte-os
      punpckhwd xmm1, xmm13   ; egész konvertálása
      punpcklwd xmm10, xmm13  ; 4 byte-os egésszé
      punpckhwd xmm3, xmm13
      cvtdq2ps  xmm0, xmm9    ; Majd floating-gá
      cvtdq2ps  xmm1, xmm1
      cvtdq2ps  xmm2, xmm10
      cvtdq2ps  xmm3, xmm3
      sqrtps    xmm0, xmm0    ; Négyzetgyök vonások
      sqrtps    xmm1, xmm1
      sqrtps    xmm2, xmm2
      sqrtps    xmm3, xmm3
      movups    [rsi+rbx*4], xmm0
      movups    [rsi+rbx*4+16], xmm1
      movups    [rsi+rbx*4+32], xmm2
      movlps    [rsi+rbx*4+48], xmm3

      add     rbx, 14        ; A 14 Sobel érték mentése
      cmp     rbx, rdx
      jl      $ujabb_oszl

      sub     rax, 1         ; a sorok számának csökkentése
      add     r8, rdx
      add     r9, rdx
      add     r10, rdx
      add     rsi, [rsp+.outpt]
      cmp     rax, 0
      jg      $ujabb_sor
      ; az elmentett regiszterek visszaállítása
      ret
```

12. ábra: A Sobel operátor megvalósítása x64 assembly-ben

Láthatjuk, hogy a kód hatékonyságát, legnagyobb részt a SSE utasítások adják. Akárcsak C-ben itt is különös gondot kell fordítani a típuskonverziókra, hiszen egyes műveleteket – pl. négyzetgyökvonás - csak megadott típusokkal vagyunk képesek elvégezni. Mindamellett, ha végignézzük a kódot, akkor láthatjuk, hogy mindössze két helyen található benne feltételes ugró utasítás. Ez azt jelenti, hogy a program több, mint 95%-a várakozó ciklusok nélkül tud végigmenni a processzor pipeline-ján, mert nincs szükség fölösleges várakozó ciklusok beiktatására. A kódban található két feltételes ugró utasítás közül a külső ciklus szervezése láthatóan úgy van megoldva, hogy a cmp utasításnak már ne kelljen várnia az rax regiszter értékére, hanem az az utasítás végrehajtása előtt már elérhető legyen, hiszen négy utasítással korábban már az utolsó olyan művelet, amely használta ezt a regisztert, már befejeződött. Várakozó ciklust csak a cmp utasítás eredményének elérése érdekében kell beiktatni a processzornak. A belső ciklusnál ez a gondolatmenet nem lett kialakítva.

4.3 Egy kivétel

A SIMD utasítások fejlődésétől azt várnánk, hogy a később rendszerbe állított utasítások gyorsabbak, mint az előző verzió ugyanolyan vagy hasonló műveletei. Erre szeretnénk egy ellenpéldát mutatni.
A következő feladatban az AVX és az SSE utasításuk hatékonyságát vizsgáljuk. Azt feltételeztük, hogy a később megjelenő utasítások gyorsabbak, hatékonyabbak, mint a régiek, vagyis az egymásnak megfeleltethető utasítások közül a később megjelentek a gyorsabbak. A kitűzött feladat egyszerűnek látszik. Két 8 kB-os memóriaterületről beolvasott adat páronkénti összegét kell kiírni a memóriába.

Ez kétszer 8 kB-nyi adat beolvasását jelenti a memóriából, majd végül 8 kB adat kiírását a memóriába. A teszt egymilliószor történő elvégzéséhez három különböző kód áll a rendelkezésünkre. A teszteléshez elkészített rutin, az egyszerűség kedvéért csak beolvas két értéket két regiszterbe és az egyiket mindenféle egyéb művelet beiktatása nélkül visszaírja a memóriába. A program így természetesen használhatatlan, de tartalmazza a legfőbb memória műveleteket. A 13. ábrán bemutatott első kód, egy hagyományos x64-es assembly utasításokkal megírt program. Az egy ciklusban feldolgozandó 64 byte-nyi információt nyolc blokkban dolgozza fel. Az eredmény az rax regiszterben képződik. Az első 8 kB-os adatterület kezdő címe az r8-as regiszterben, míg a második adatterületé az r9-es regiszterben található. Az eredményt az r10-es regiszterben megadott báziscímre írjuk. Az aktuális 64 byte-nyi adat címe az r11-es regiszterbe kerül, melynek értéke 0 és 1023 között változik.

```
mov rax, QWORD PTR [r8+r11*8]
mov rdx, QWORD PTR [r9+r11*8]
; itt történne meg az adatfeldolgozás
mov QWORD PTR [r10+r11*8], rax
```

13. ábra: Az I/O műveletek x64 assemblyben

A második utasítás után jönnének a beolvasott adatokon elvégzendő műveletek. Most, a 2. utasítás helyett akár az

```
add rax, QWORD PTR [r9+r11*8]
```

is használhatnánk, de az azonos típusú műveletek alkalmazása miatt most csak memória műveleteket használunk.
A továbbfejlesztett második kód már SSE2 utasításokat tartalmaz. Ahogy azt a 14. ábra is mutatja, ebben a programban, a 128 bites regiszterek miatt, a ciklusszám már az eredeti felére esik.

```
movdqa xmm0, XMMWORD PTR [r8+r11*16]
movdqa xmm1, XMMWORD PTR [r9+r11*16]
; itt történne meg az adatfeldolgozás
movdqa XMMWORD PTR [r10+r11*16], xmm0
```

14. ábra: Az I/O műveletek SSE2 utasítások felhasználásával

Azt nem várhatjuk, hogy a második program hatékonyága megduplázódik, mert bár a ciklusszám ugyan az eredeti felére esik vissza, nem szabad elfelejteni, hogy a beolvasott adatok nagysága viszont megnő.
A harmadik, AVX utasításokat tartalmazó programrészlet a 15. ábrában látható.

```
vmovdqa ymm0, YMMWORD PTR [r8+r11*32]
vmovdqa ymm1, YMMWORD PTR [r9+r11*32]
; itt történne meg az adatfeldolgozás
vmovdqa YMMWORD PTR [r10+r11*32],ymm0
```

15. ábra: Az I/O műveletek AVX utasítások felhasználásával

Ahogy az előbb sem, most sem duplázódik meg a harmadik program hatékonyága, ugyanazon okok miatt, mint az előző esetben. A várakozásnak megfelelően a leglassabb kódnak az elsőt, másodiknak a másodikat, és leggyorsabbnak az utolsó kódot vártuk. A három különböző megközelítésű teszt végeredménye azonban furcsa értékeket mutatott. A várttal ellentétben azonban a teszt, ahol az átlagos gépi ciklusok számát határoztuk meg, az 1. táblázatban foglalt eredményeket mutatta.

Vajon miért lesz lassabb az AVX változat, mint a régebbi SSE2-s? Arra esetleg lehetett számítani, hogy a ciklusfelezés és a memóriaolvasás megegyező nagyságú, de ellentétes hatása miatt az AVX változat csak egy kicsit lesz gyorsabb, mint az SSE. De hogy lassabb legyen, még ha csak egy kicsit is, erre nem számítottunk. Esetleg az YMM regiszterek használata okozza ezt az időtúllépést?
A kérdésekre nehezen találtunk válaszokat.

Eljárás	Gépi ciklus/64 byte
x64	60
SSE2	34
AVX	36

1. táblázat: A gépi ciklusok száma egységi adatmennyiségre vetítve

A Sandybridge szerint a 256 bites AVX utasítások egy része két 128 bites utasításban végzi el és tárolja a kettévágott adatokat. Mivel egy művelet valójában egy kettős ciklust tartalmaz, ezért szükség van a második ciklusban arra, hogy a művelet, hogy törölje vagy alaphelyzetbe állítsa a portokat vagy a végrehajtó egységeket, így nem számíthatunk arra, hogy ez a verzió, ezeket az utasításokat sokkal gyorsabb végre tudja hajtani.
A másik lehetőség, amit figyelembe vehetünk az az, hogy az Intel Optimization Manual-ja szerint a bázis + index + ofszet címzést 128 bites regiszterek mellett a processzor 6 gépi ciklus alatt hajtja végre, míg ugyanezt az utasítást 256 bites regiszterek esetén már 7 ciklus (2-8 táblázat az Intel Optimization Manual). Ebben az esetben a hosszabb várakozási idő valószínűleg azt jelenti, hogy a processzor arra vár, hogy biztonságosan felépüljön, mondjuk a pipeline előrejelzések okozta várakoztatásokból, vagy megszakításokból, melyek biztos van némi hatása. Mivel ezekre az elképzelésekre nincs pontos gyári adat, így csak arra lehet következtetni, hogy a processzor biztonságos működtetése okozhatja ezeket az eltéréseket.

ÖSSZEFOGLALÁS

Az előző fejezetekben kísérletet tettünk a képi adatfeldolgozás gyorsítása érdekében bevethető lehetőségek bemutatására. Az egyik ilyen eszköz a magasszintű és a gépközeli nyelvek kombinálása. Természetesen nem kell egy közlekedési megfigyelő rendszer összes rutinját gépközeli nyelven megírni, de azokat a kulcsfontosságú rutinokat, amelyek a legtöbbször kerülnek működésbe és/vagy nagyon fontos a gyorsaságuk, mindenképpen érdemes. Bemutattunk néhány lehetőséget, amely segíthet ebben a munkában. A SIMD utasítások használata, ahogy azt a 4. fejezetben olvashattuk is, jelentősen meggyorsíthatja az adatfeldolgozás folyamatát. Bár az itt bemutatott példák az Intel Architecture-ákra használhatók, maguk a lehetőségek, mint például az

elágazásmentes kódok használata, bármilyen hardver környezetben használható.

FELHASZNÁLT IRODALOM

Kanhere, N.K., Birchfield, S.T., Sarasua, W.A., Whitney, T.C., (2007). *Real-Time Detection and Tracking of Vehicle Base Fronts for Measuring Traffic Counts and Speeds on Highways,* in Transportation Research Board Annual Meeting, Washington, D.C., January 2007. TRB Paper Number: 07-3308

Kun, A.J., Vamossy, Z., (2009). *Traffic monitoring with computer vision*, Proc. 7th Int. Symp. Applied Machine Intelligence and Informatics (SAMI 2009)

Kusswurn, D. (2014). *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*, Apress, 2014. ISBN13: 978-1-484200-65-0

Max, G., (2012). *Közlekedési szabálytalanságok*, IFFK 2012, Budapest, ISBN 978-963-88875-2-8

Max, G. (2014). *Gépjárműtípus felismerés beépített SW segítségével*, IFFK 2014, Budapest, ISBN 978-963-88875-2-8

Seyfarth, R., (2014). *Introduction to 64 Bit Assembly Programming for Linux and OS X*, CreateSpace Independent Publishing Platform, 2014. ISBN: 978-1484921906

Seyfarth, R., (2014). *Introduction to 64 Bit Assembly Programming*, CreateSpace Independent Publishing Platform, 2014. ISBN: 978-1484921968

Uke, N.J., Thool, R. (2012). *Moving Vehicle Detection for Measuring Traffic Count Using OpenCV*, in Proc. 4th International Conference on Conference on Electronics Computer Technology (ICECT 2012)

[Int01]https://www.mikekohn.net/stuff/image_processing.php

[Int02]http://www.codeproject.com/Articles/36144/Optimized-Image-Inversion-Using-SSE

[Int03]http://stackoverflow.com/questions/10329903/efficient-complex-arithmetic-in-x86-assembly

[Int04]http://stackoverflow.com/questions/10581451/detection-of-rectangular-bright-area-in-a-image-using-opencv

[Int05]http://stackoverflow.com/questions/13975546/avx-vmovdqa-slower-than-two-sse-movdqa

[Int06]http://stackoverflow.com/questions/14656010/using-opencv-to-detect-parking-spots

[Int07]http://stackoverflow.com/questions/1715224/very-fast-memcpy-for-image-processing

[Int08]http://stackoverflow.com/questions/17815687/image-processing-implementing-sobel-filter

[Int09]http://stackoverflow.com/questions/34337013/bitmap-image-processing-with-masm32

[Int10]https://software.intel.com/en-us/articles/introduction-to-x64-assembly

FÜGGELÉK

Korreláció számítás rutinja assembly-ben SSE kódok alkalmazásával.

```
segment .text
        global corr

; Regiters used: rdi, rsi, rdx, rcx, r8, r9
;       rdi:  x array
;       rdi:  y array
;       rcx:  loop counter
;       rdx:  n
;       xmm0: 2 parts of sum_x
;       xmm1: 2 parts of sum_y
;       xmm2: 2 parts of sum_xx
;       xmm3: 2 parts of sum_yy
;       xmm4: 2 parts of sum_xy
;       xmm5: 2 x values - later squared
;       xmm6: 2 y values - later squared
;       xmm7: 2 xy values
corr: xor       r8, r8          ; r8=0
      mov       rcx, rdx        ; rcx=n
      subpd     xmm0, xmm0      ; xmm0=0
      movapd    xmm1, xmm0
      movapd    xmm2, xmm0
      movapd    xmm3, xmm0
      movapd    xmm4, xmm0
      movapd    xmm8, xmm0
      movapd    xmm9, xmm0
      movapd    xmm10, xmm0
      movapd    xmm11, xmm0
      movapd    xmm12, xmm0
ciklus:
      movapd    xmm5, [rdi+r8]  ; mov x
      movapd    xmm6, [rsi+r8]  ; mov y
      movapd    xmm7, xmm5      ; mov x
      mulpd     xmm7, xmm6      ; xy
      addpd     xmm0, xmm5      ; sum_x
      addpd     xmm1, xmm6      ; sum_y
      mulpd     xmm5, xmm5      ; xx
      mulpd     xmm6, xmm6      ; yy
      addpd     xmm2, xmm5      ; sum_xx
      addpd     xmm3, xmm6      ; sum_yy
```

```
addpd       xmm4, xmm7      ; sum_xy
movapd  xmm13, [rdi+r8+16] ; mov x
movapd  xmm14, [rsi+r8+16] ; mov y
movapd      xmm15, xmm13   ; mov x
mulpd       xmm15, xmm14   ; xy
addpd       xmm8, xmm13    ; sum_x
addpd       xmm9, xmm14    ; sum_y
mulpd       xmm13, xmm13   ; xx
mulpd       xmm14, xmm14   ; yy
addpd       xmm10, xmm13   ; sum_xx
addpd       xmm11, xmm14   ; sum_yy
addpd       xmm12, xmm15   ; sum_xy
add         r8, 32
sub         rcx, 4
jnz         ciklus
addpd       xmm0, xmm8
addpd       xmm1, xmm9
addpd       xmm2, xmm10
addpd       xmm3, xmm11
addpd       xmm4, xmm12
haddpd      xmm0, xmm0      ; sum_x
haddpd      xmm1, xmm1      ; sum_y
haddpd      xmm2, xmm2      ; sum_xx
haddpd      xmm3, xmm3      ; sum_yy
haddpd      xmm4, xmm4      ; sum_xy
movsd       xmm6, xmm0      ; sum_x
movsd       xmm7, xmm1      ; sum_y
cvtsi2sd    xmm8, rdx       ; n
mulsd       xmm6, xmm6      ; sum_x*sum_x
mulsd       xmm7, xmm7      ; sum_y*sum_y
mulsd       xmm2, xmm8      ; n*sum_xx
mulsd       xmm3, xmm8      ; n*sum_yy
subsd       xmm2, xmm6 ; n*sum_xx-sum_x*sum_x
subsd       xmm3, xmm7 ; n*sum_yy-sum_y*sum_y
mulsd       xmm2, xmm3      ; denom*denom
sqrtsd      xmm2, xmm2      ; denom
mulsd       xmm4, xmm8      ; n*sum_xy
mulsd       xmm0, xmm1      ; sum_x*sum_y
subsd       xmm4, xmm0 ; n*sum_xy-sum_x*sum_y
divsd       xmm4, xmm2      ; correlation
movsd       xmm0, xmm4      ; need in xmm0
ret
```

ahol a CVTSI2SD (Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value) utasítás a 4 byte-os egész számokat alakítja át dupla pontosságú skalár lebegőpontos valós számokká.